Dynamic Tuning and Overload Management of Thread Pool System

Faisal Bahadur, Arif Iqbal Umar, Fahad Khurshid
Department of Information Technology, Hazara University
Mansehra, K.P.K. Pakistan

***Abstract*—Distributed applications have been developed using thread pool system (TPS) in order to improve system performance. The dynamic optimization and overload management of TPS are two crucial factors that affect overall performance of distributed thread pool (DTP). This paper presents a DTP, that is based on central management system, where a central manager forwards client's requests in round robin fashion to available set of TPSs running in servers. The dynamic tuning of each TPS is done based on request rate on the TPS. The overload condition at each TPS is detected by the TPS itself, by throughput decline. The overload condition is resolved by reducing the size of thread pool to previous value, at which it was producing throughput parallel to the request rates. By reducing the size of thread pool on high request rates, the context switches and thread contention overheads are eliminated that enables system resources to be utilized effectively by available threads in the pool. The result of evaluation proved the validity of proposed system.**

Keywords*—Distributed system; distributed thread pool; thread pool system; performance; overload management

I. INTRODUCTION

The ever-growing expansion of Internet and World Wide Web demands scalable services that must be performance efficient and highly available. The prominent progression of internet's user doubles internet traffics every two or three months. For example, OSN sites such as LinkedIn, Flickr, Myspace, Twitter and Facebook provide facilities to over half a billion users at the same time [1]. The OSN's not only provide basic communication capabilities but also provide other services by third party applications e.g. sharing documents, sending virtual gifts, or gaming. Number of third-party applications run by the Facebook are over 81,000 [1]. There is a profound impact of these third-party applications on the application server's scalability and performance thus results in additional traffic. For example, when Facebook launched its developer platform, the traffic increased by 30% in a week after launching [2], while in case of Twitter the traffic increased by a factor of 20 after launching its API [3]. Also, the variations in demand go to extreme levels in some internet services that cause overload condition and needs special attention to manage server-side resources. In order to deal with these complexities, internet services are provided by distributed application servers [4], that are responsible of providing run time services to applications, where these applications service demands of many concurrent users.

At present, distributed systems have been implemented in almost all domains including telecommunication, defense, industrial automation, financial services, entertainment, government and e-commerce. And that is why, the requirements of complexity management, scaling and overload management are increasing day by day. As discussed earlier, distributed systems handle heavy workloads, where client's requests are incoming from a remote source through some network protocol. These heavy workloads are handled by distributed systems through extremely concurrent design configurations that are implemented as middleware. The performance of distributed systems is dominated by middleware that provide different functionalities, e.g. multithreading policies, remote communication mechanisms, persistence services and transaction management etc. It is the middleware that makes distributed system scalable, highly available and highly performant [5]. Some remarkable examples of middleware services for distributed systems are middleware of Distributed Object Computing (such as CORBA, SOAP, RMI) Component middleware (such as .NET, Java Beans), Message Oriented Middleware (such as Java Message Queue, BEA's WebLogic MessageQ) etc.

One of the most important performance related feature of any middleware service in distributed systems is concurrency control that handles multiple concurrent requests. Two most commonly used concurrency models are Thread Pool System (TPS) and event driven model (EDM).

As compared to TPS, EDM is more performance efficient, but at the same time it is much complicated and challenging to implement than TPS [6]. The most challenging task in EDM is to handle scheduling and assembling of events [7]. Moreover, EDM leads to enormous cascading callback chains [8]. As compared to EDM, TPS offers more solid structuring constructs for concurrent servers by means of threads that are light weight and represent work from the perception of the task itself [9,10]. Moreover, TPS avoids resource thrashing and overheads of thread creation and destruction [11]. Some examples of TPS in middleware for distributed systems include .NET thread pool [12], Java Message Queue Thread Pool [13].

A typical TPS contains a request queue, a pool of threads (workers) and dynamic optimization algorithm (DOA) that optimizes pool size, as shown in Fig.1.

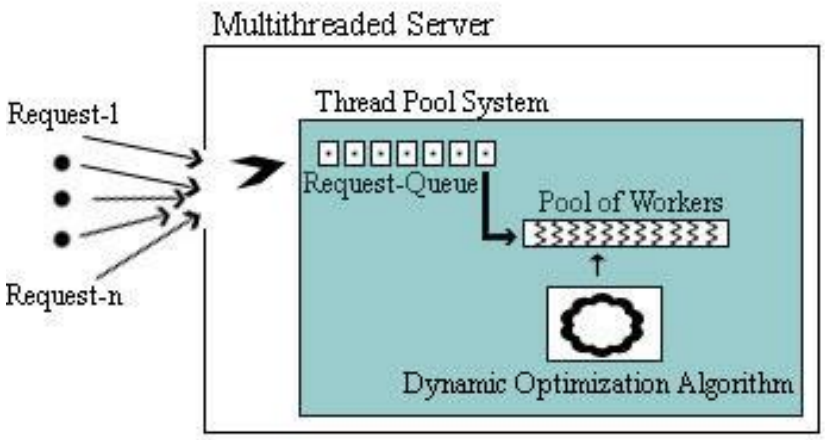

Fig. 1. Conceptual Model of Thread Pool System Embedded in a Server.

Request queue stores incoming client's requests. Worker threads in the pool fetches and executes these requests. These worker threads in the pool are recycled (instead of being destroy) to process next client's request from queue. The re-spawning and recycling of worker threads avoids thread creation and destruction costs, but, under a heavy load scenario, additional threads must be dynamically created and inserted inside pool to cope with the load. The DOA component of TPS is responsible to decide the quantity of extra threads. It is a challenging task of DOA to maintain an optimal pool size on run time, in order to produce better response times and maximum throughput so that quality of service can be maintained. If thread pool size is beyond an optimal limit, then it increases thread context switches and thread contention (on shared resource), that ultimately provides poor performance. On the other hand, if pool size is smaller than an optimal limit then it results in poor response time and throughput. Handling this tradeoff on run time is essential to achieve best performance. Optimizing thread pool size by DOA is not an exact science and it can be performed on the basis number of parameters and factors.

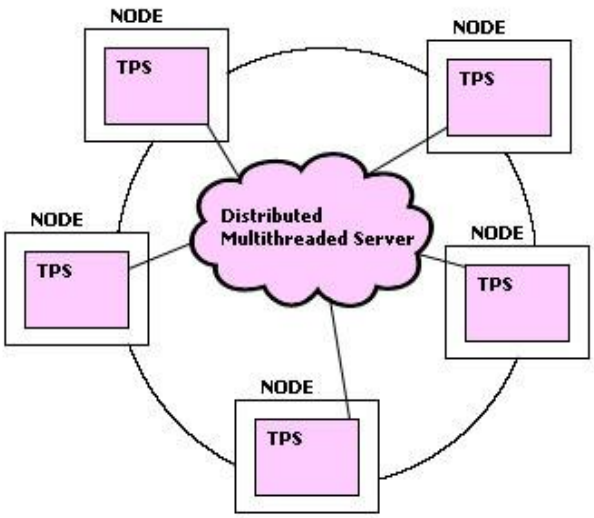

Fig. 2. Conceptual Model of Distributed Thread Pool System.

The variety of target servers where TPSs are installed makes DOA more challenging, as there are varied characteristics of the deployment system with a diverse nature of tasks. Because of this reason, TPS has been evolved from single-pool to multi-pool and from multi-pool to DTP. DTPs are designed for distributed systems where they are horizontally scaled over number of nodes available on the network as shown in Fig. 2.

In DTPs, overload monitoring of each TPS running in a server node is a crucial factor to avoid unsuitable large quantity of threads in the pools, so that overheads of thread contention and context switches may be reduced. Moreover, metric for detecting overload condition must be selected carefully in order to gain maximum performance.

This work is based on our previous work [14], in which, we have presented a distributed framework of TPS called distributed frequency based thread pool (DFBTP), where each server node has its own TPS, that is tuned on the basis of request arrival rate, and the load on each node is balanced by a round robin strategy in order to fairly distribute the load among available TPSs. In this paper, we have extended DFBTP by presenting overload control based distributed thread pool (OCBDTP), whose overload control mechanism tackles overload condition by detecting throughput fall, in case of high request frequencies. In such a case, thread pool size is restored to previous appropriate size, where it was running normally.

The rest of the paper is organized as follows. Section 2 presents literature review. The design of is presented in section 3. The validation of proposed system is detailed in section 4, and finally conclusion is given in section 5.

II. Related Work

A mathematical model for dynamic optimization of TPS is presented in [15], that forms a relationship among system load, pool size and the costs associated with thread creation, destruction and maintenance. However, the estimated optimal thread pool size might be inaccurate, since accurately estimating the time for thread creation and thread context switching is difficult in practice.

CPU-utilization based TPS is presented in [16], where dynamic optimization scheme increases pool size when CPU utilization decreases and vice versa. It defines a lower threshold and upper threshold variables for CPU utilization, and an update function repeatedly optimizes pool size on the basis of these threshold variables. This scheme however can't be used for I/O bound applications.

TPS presented in [17] performed dynamic optimization by calculating average idle time (AIT) of queued requests. This scheme increases pool size if AIT is increasing. However, the uses of too many thresholds in dynamic tuning algorithm have no justification and can affect the performance.

A prediction-based scheme is utilized in [18], in order to set a pool size in advance for future use by Gaussian distribution. These predictions might be inaccurate due to synchronization overhead.

Exponential moving averages were utilized in [19], in order to predict pool size in advance. This scheme suffered from creating redundant threads.

Thread borrow scheme was used in an application server [20], that utilized multiple thread pools, but management of multiple pools is itself a cost-effective operation.

A model fuzzing approach was used in [21], to optimize pool size. Number of parameters and constraints were utilized for optimizing pool size which was too complex to quickly make a conclusion, hence it is not suitable for the system having frequent state change.

TPS presented in [22], used response coefficient to optimize pool size. However, this metric is normally affected by different run time parameters.

By extending the work of [18], the trends of time series in exponential moving averages were analysed in [23], in order to avoid redundant threads. This scheme however suffered from creating lacking threads.

A theoretical framework of distributed thread pool was presented in [24], that is governed by software agents that can dynamically add and remove threads in the pool based on load conditions.

A design of hierarchical thread pool executor based on non-blocking queue was presented in [25]. This TPS was presented for java-based DSMs running on cluster. This TPS served as an outclass alternative to Jackal's thread pool executor which is based on blocking queue. This design was targeted to only DSM systems.

The dynamic optimization of TPS in [26] is performed by performance data of threads associated with system resource usage. The calculated value is gradually updated and compared with old values over time. This trend is analysed over time to perform dynamic optimization.

It is argued in [27], that response times of a TPS-based application suffer when it utilizes more threads than required, as it increases context switching overhead. And that is why he established an inverse proportional relationship of response time to pool size. This TPS gradually decreases pool size on high response times, until system's stability.

A divide and conquer strategy is used in [28] that divides the tasks into subtasks by a pipeline based technology that allows the sub-tasks to run in parallel that reduced the computational cost. But dynamic division of tasks into subtasks was challenging.

Thread pool system presented in [29] is optimized based on application level metric called thread utilization. On high thread utilization, thread pool size is increased and vice versa.

A multiple pool approach is used in [30], where each pool is reserved to process requests having specific service time. In this way requests having large service times are separated from requests having small service times, hence avoiding large requests to block small ones in order to occupy all threads in the pool. However, a large variation in service times of incoming requests results in large number of pools that increases pool management overhead.

In [31], a middleware for CPS-systems is presented which utilized a linear approach to tune its thread pool. Thread pool is resized on the basis of increase in request rate. The system uses unbounded DTS that is only suitable for CPS systems.

A dynamic framework is offered in [32] for n-tire application that is running in the cloud-based system. At each tier, thread pool is tuned by queuing laws and system-level metrics.

In our previous work [14], we have presented DFBTP that was designed for distributed applications, where each node has its own thread pool. The size of each thread pool is optimized by frequency of incoming requests and load is equally distributed among all nodes by a round robin scheme. This scheme creates threads in the pool based on request arrival rate that can be very large on heavy load situations that may result in very large pool size that effects performance.

III. MOTIVATION

Each TPS in DFBTP [14] keeps pool size equal to the request arrival rate. In case of high request arrival rate, DFBTP creates a very large pool size that effects performance. Fig. 3 is an illustrative diagram, that demonstrates the problem of throughput degradation due to worse pool size on heavy load.

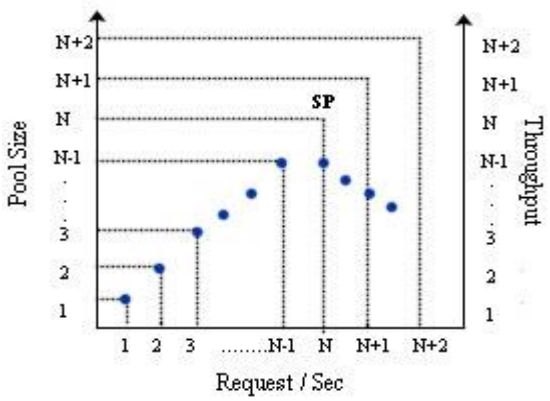

Fig. 3. Throughput Starts Falling on Saturation Point.

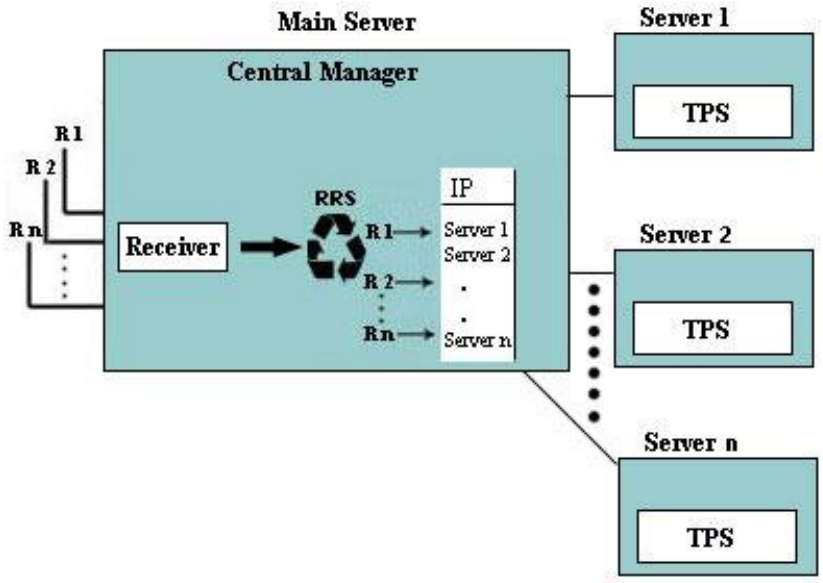

Fig. 4. Architecture of Distributed System having Frequency-based Thread Pools on Slave Servers.

X-axis represents request arrival rate, whereas y-axis consists of multiple axis. PoolSize axis represents number of threads in the thread pool, and Throughput axis represents number of requests completed per second. Throughput is represented by blue dots. We can see in Fig.1, that pool size is equal to the request arrival rate at every moment. Also, throughput is gradually increasing for some moments and equal to the request arrival rate. When request arrival rate is N, the system kept pool size equal to N also, however, throughput decreased and turned into N-1. This point is called saturation point (i.e. overload condition), where throughput started falling, because, DFBTP continuously increasing pool size on next successions and due to very large pool size, system is busy in context switch overhead and thread management overhead. System resources are busy in managing thread context switches and thread contention, instead of doing useful work. The motive of proposed scheme is to tackle saturation point and react accordingly.

IV. Material and Method

In this section we discuss in detail, the design and implementation of OCBDTP. First, we will discuss the distributed architecture of our proposed system, and next we will describe the design of proposed TPS, followed by overload control mechanism.

A. Proposed Distributed System Architecture

OCBDTP consists of a central manager (CM) component that is running in the main server, and one or more TPS running in server nodes, as shown in Fig. 4. System initialization starts from CM that runs in the main server first and waits for the TPS to connect with it. When a TPS starts on a server node in the network, it connects to the CM, that in turn stores the IP address of corresponding server in a list and starts its Receiver thread that will accept client's requests and handover to round robin scheduler (RRS). RSS is a load balancer component that equally distributes the load on all available TPS by iterating over the list that contains IP addresses of servers. CM is now ready to accept client's request. On request arrivals, Receiver will receive requests and forward it to the RRS, that in turn distribute these requests to TPSs. Each TPS is responsible to accept requests and process these requests.

B. System Architecture of TPS

The architecture of proposed TPS is shown in Fig. 5. When a TPS starts in the server node, it initializes three detector threads named Request Rate Detector (RRD), Throughput Detector (TD) and Saturation Detector (SD). Next, it connects to the CM through Connector component. On arrival of first request, TPS starts detector threads that perform their tasks after every second until TPS is running.

A counter is incremented on every request arrival. Each request arrived at TPS is first inserted into request queue and picked up by any available thread inside pool that will process request and store it as a response in response queue. RRD activates after every second, reads the value of counter, stores it as a current request frequency and sets the value of counter again to zero in order to detect the next request arrival rate. RRD maintains request rates of current and previous phases. These request rates are later read by SD in order to adjust pool size equal to the request frequency if throughput is not falling. TD activates after every second, counts and de-queues all the responses from response queue. The counts of responses are stored in Throughput object. The responses are sent back to the CM, that in turn sends these responses back to the client. TD maintains throughputs of current and previous phases.

SD also activates after every second that performs two jobs. First, it tunes thread pool size based on request rates. Second, it periodically measures throughput of TPS in order to detect overload condition. In case of overload condition SD reacts accordingly that is discussed in the next section.

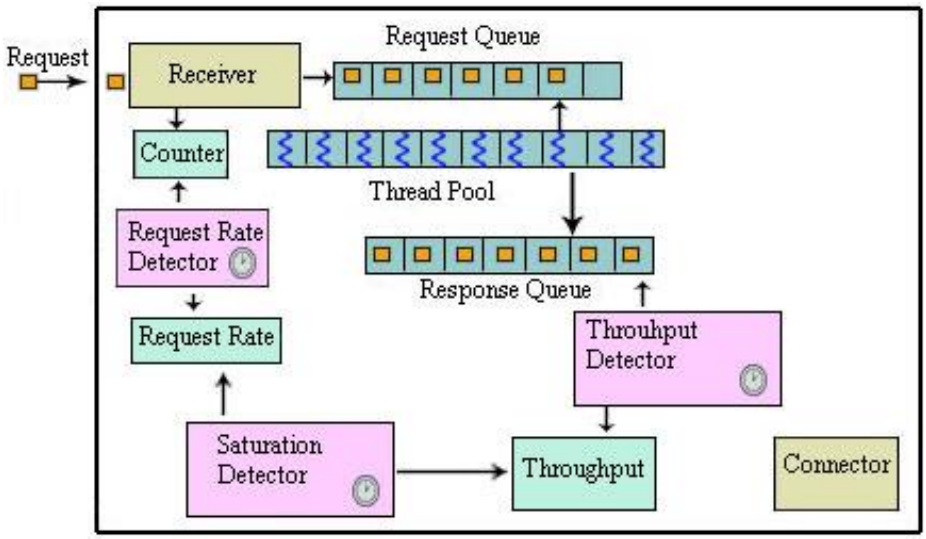

Fig. 5. Architecture of Proposed Thread Pool System.

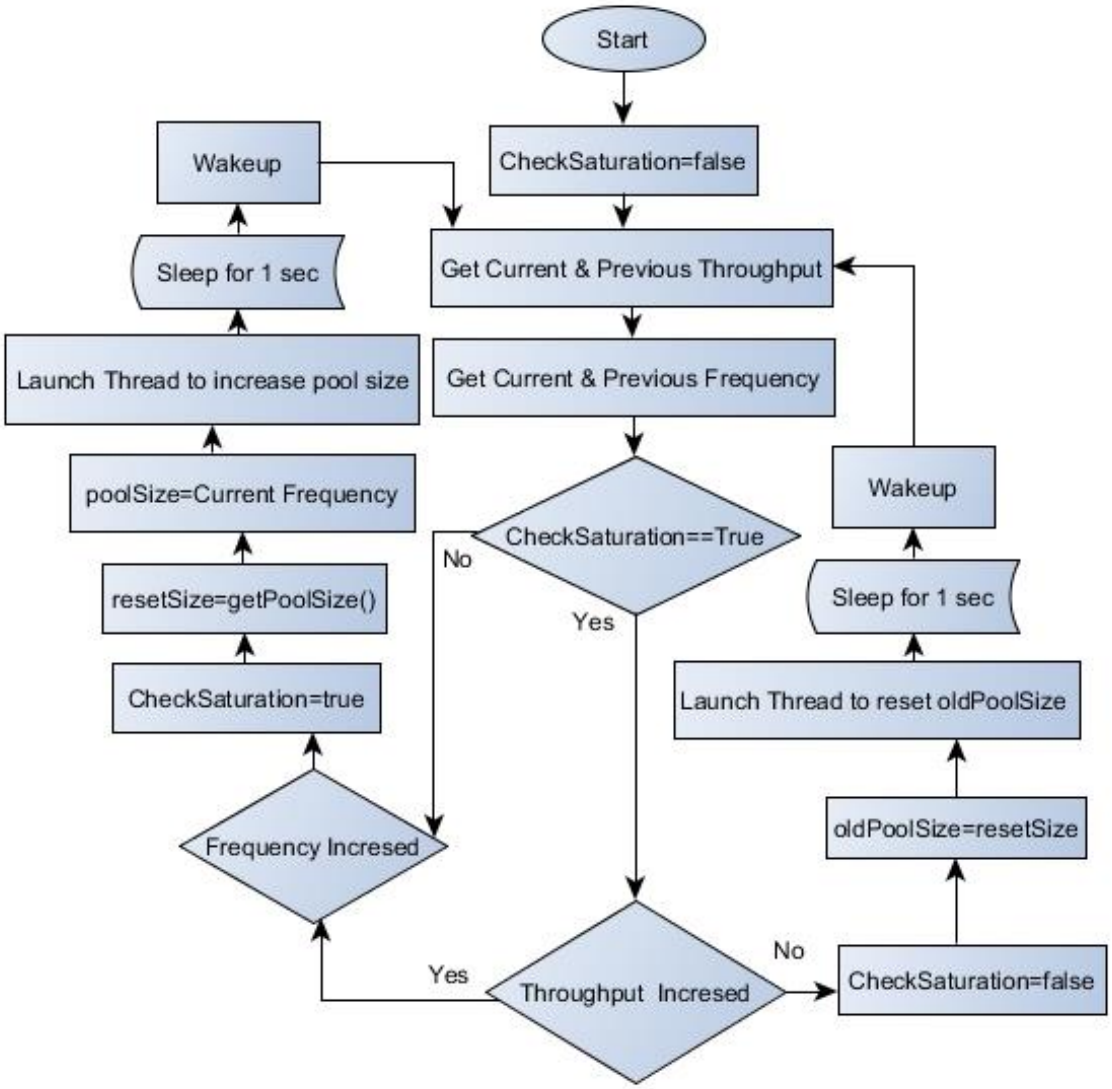

Fig. 6. Flowchart Diagram of Saturation Detection and Avoidance Mechanism.

C. Overload Detection and Control

Fig. 6 is a working flow chart diagram of SD, that dynamically sets an appropriate pool size of thread pool that can avoid thread context switch and contention overheads. And this is done by detecting an inappropriate pool size at which throughput starts falling. SD algorithm first collects values of throughout and frequency regarding current and previous phases. Then, it checks for increase of request rate; if it is increasing, then it sets pool size equal to the request rate by running a separate thread in the background and repeats itself after one second. In the next pass it first checks for throughput gain. Since quantity of threads was increased in the last phase, that means more throughput should be there, and throughput should be equal to the size of thread pool per second. If throughput is improved and equals to the number of threads in the pool, then the size of thread pool is considered to be appropriate and it is saved for later use in case of overload, so that, an inappropriate size can be restored to saved one. Next, it again checks for current request rate; performs corresponding steps; and repeat after one second. In case throughput is not improved, it means that pool size was not set to an optimal value in the last phase, so it resets pool size again to the saved one (by running a separate thread in the background) where TPS was running normally, i.e. giving throughput equal to the request rate. On high request rate, SD first keeps pool size high, and watches over throughput. However, due to large number of threads in the pool beyond the capacity of system, the system turns into a state, where it is busy in managing thread context switches and thread contention, instead of doing useful work by threads in the pool. And that is why, its throughput starts falling initially. However, SD detects this overload condition, and resets pool size to the previous phase's size where there were no overheads of thread context switches and contention. Restoring pool size eliminates overheads of thread context switches and contention and system resources are effectively utilized by available threads that again restores system performance.

V. Results and Discussion

In order to validate proposed thread pool system, we simulated it using a java-based toolkit named Pool-Runner[1] that can embed a DTP inside its server tier for performance testing. Its client tier has a load generation engine that displays results in form of graphs. We performed simulation on a network with three machines. First machine has a client tier of toolkit, second machine is used as a main server running CM and third machine is used as a slave running a TPS.

We use static workload for this test. The static workload is simulated by StaticTask object (available in toolkit) that simulates 1kb file by sleeping for 100 milliseconds (approximately). Load is first generated by poisson distribution with the rate of 1000 request/sec as shown in Fig. 7. After every minute we increased the load. As shown in Fig. 7 that the load is 2000 request/sec in second minute, and 3000 request /sec in third minute.

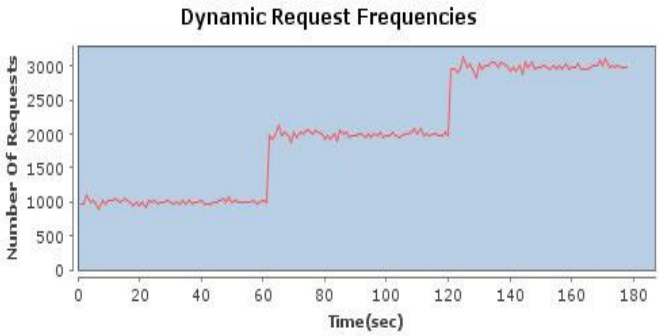

Fig. 7. System Load Used for Simulation.

Fig. 8 shows comparative analysis of thread pool size between DFBTP and OCBDTP, where DFBTP kept its pool size equal to request rate all the time, whereas OCBDTP kept its pool size equal to the request rate till 2 minutes, however, pool size is restored to previous appropriate size that was in the second minute. And this is because of throughput falling in 3rd minute. When request rate turned into 3000 request/sec in 3rd minute, OCBDTP detected saturation in the system, that can be seen in Fig.9 that shows that throughput started falling initially but raised up again when pool size is set to an optimal level.

Fig. 9 shows throughput per second, where DFBTP sustained its throughput in first two minutes, as average throughput is equal to the request rate, however in 3rd minute, its throughput is continuously falling because its pool size is very large that is creating thread management (context switches and contention) overheads. OCBDTP also sustained its throughput till 120 seconds, however, when request rate raised up to 3000 requests/sec, its throughput started falling initially, that is recovered by SD, and pool size is set to an optimal value that caused throughput to sustain again.

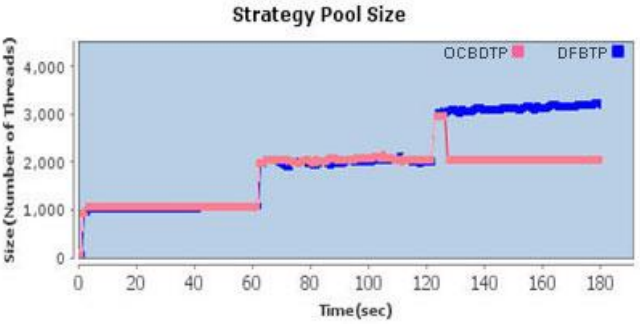

Fig. 8. Comparative Analysis of Pool Size.

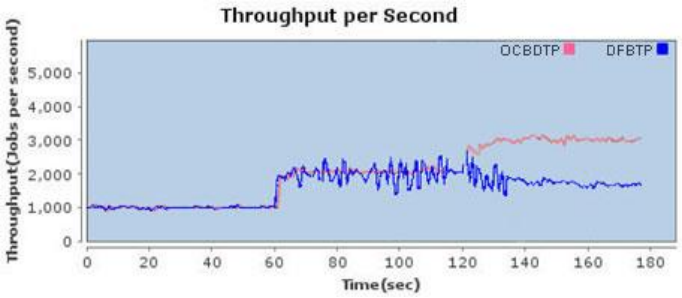

Fig. 9. Comparative Analysis of Throughput Per Second.

[1] http://www.jpoolrunner.net

VI. Conclusion

This paper has presented a distributed thread pool system, that is based on central manager, that forwards client's requests (in round robin fashion) to all instances of thread pools running in the distributed environment. The overload condition at each TPS is detected by throughput decline and resolved by reducing and restoring pool size to previous appropriate value, at which the system was stable. The overload control (by reducing pool size) eliminates thread management overhead, that enables system resources to be used effectively by threads in the pool. Proposed system prevents pool size to increase beyond an appropriate level, hence avoids overheads of thread context switching and contention, hence increases system performance. Proposed system results in more throughput gain. In the future, we will tune thread pools by software agents having artificial intelligence capabilities that would be used on networks and clusters for automatic control of distributed system resources.

References

[1] A. Nazir, S. Raza, D. Gupta, C-N. Chuah, B. Krishnamurthy, "Network level footprints of facebook applications, "In Proceedings of the 9th ACM SIGCOMM conference on Internet measurement, pp. 63-75, 2009.

[2] A. Nazir, S. Raza, C.-N. Chuah, "Unveiling facebook: A measurement study of social network based applications, "In Proc. Internet Measurement Conference (IMC), 2008.

[3] B. Krishnamurthy, P. Gill, M. Arlitt, "A few chirps about twitter," In Workshop on Online Social Networks, 2008.

[4] N.Aghdaie , Y.Tamir, " Fast transparent failover for reliable web service," InInternational Conference on Parallel and Distributed Computing and Systems, pp. 757-762, 2003.

[5] G. Denaro, A. Polini, W. Emmerich, "Performance testing of distributed component architectures," In Testing Commercial-off-the-Shelf Components and Systems, Springer, Berlin, Heidelberg, pp. 293-314, 2005.

[6] M. Andreolini, V. Cardellini, M. Colajanni, " Benchmarking models and tools for distributed web-server systems," In: IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation, pp. 208-235, September, 2002.

[7] M. Welsh, D. Culler, E. Brewer, "SEDA: an architecture for well-conditioned, scalable internet services," ACM SIGOPS Operating Systems Review, 35(5), pp. 230-243, 2001.

[8] A. Gustaffson, "Threads without the Pain," Queue, Vol. 3, No. 9, pp. 34-41, 2005.

[9] R.V. Behren, J. Condit, E. Brewer, "Why events are a bad idea (for high-concurrency servers)," In: Proceedings of the 9th conference on Hot Topics in Operating Systems, pp.4-4, 2003.

[10] R.V. Behren, J. Condit, F. Zhou, G.C. Necula, E. Brewer, "Capriccio: scalable threads for internet services," ACM SIGOPS Operating Systems Review, Vol. 37, No.5, pp. 268-281, 2003.

[11] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, "Java Concurrency in Practice," Addison-Wesley Professional, 2005.

[12] The Managed Thread Pool https://msdn.microsoft.com/enus/library/0ka9477y(v=vs.110).aspx (accessed on 10 July 2018).

[13] ORACLE, Java System Message Queue 4.3 Technical Overview https://docs.oracle.com/cd/E19316-01/820-6424/aerck/index.html (accessed on 10 July 2018).

[14] S. Ahmad, F. Bahadur, F. Kanwal, R. Shah, "Load balancing in distributed framework for frequency based thread pools," Computational Ecology and Software, Vol. 6, No. 4, pp. 150-164, 2016.

[15] Y. Ling , T. Mullen , X. Lin, "Analysis of optimal thread pool size," ACM SIGOPS Operating Systems Review, Vol. 34, No. 2, pp.42-55, 2000.

[16] D. Robsman, "Thread optimization," US6477561 B1, 2002.

[17] D. Xu , B. Bode, "Performance Study and Dynamic Optimization Design for Thread Pool System," In Proc. of the Int. Conf. on Computing Communications and Control Technologies, pp.167-174, 2004.

[18] J. Kim , S. Han , H. Ko , H. Youn, "Prediction- based Dynamic Thread Pool Management of Agent Platform for Ubiquitous Computing," In Proceedings of UIC, pp. 1098-1107, 2007.

[19] D. Kang , S. Han , S. Yoo , S. Park, "Prediction based Dynamic Thread Pool Scheme for Efficient Resource Usage," In Proc. of the IEEE 8th Int. Conf. on Computer and Information Technology Workshop, IEEE Computer Society, pp. 159-164, 2008.

[20] T. Ogasawara, "Dynamic Thread Count Adaptation for Multiple Services in SMP Environments," IEEE International Conference on Web Services, pp. 585-592, 2008.

[21] J. Hellerstein, "Configuring resource managers using model fuzzing: A case study of the .NET thread pool," IFIP/IEEE International Symposium on Integrated Network Management (IM '09), pp. 1-8, 2009.

[22] N. Chen, P. Lin, "A Dynamic Adjustment Mechanism with Heuristic for Thread Pool in Middleware," 3rd Int. Joint Conf. on Computational Science and Optimization. IEEE Computer Society, pp. 324-336, 2010.

[23] K. Lee , H. Pham , H. Kim , H. Youn , O. Song, "A novel predictive and self-adaptive dynamic thread pool management," In: Proceedings - 9th IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 93–98, 2011.

[24] P. Martin , A. Brown , W. Powley , J. Luis , V. Poletti, "Autonomic management of elastic services in the cloud," In: Proceedings - IEEE Symposium on Computers and Communications, pp. 135–140, 2011.

[25] S. Ramisetti , R. Wankar, "Design of hierarchical thread pool executor for DSM. Second International Conference on Intelligent Systems Modelling and Simulation (ISMS), pp. 284-288, 2011.

[26] F. Muscarella, " Method and apparatus for dynamically adjusting thread pool," US8185906 B2, 2012.

[27] R.T. Gowda, "Dynamic thread pool management," US 8381216 B2, 2013.

[28] M. Chen, Y. Lu, G. Zhang, W. Zou, "Real-Time Simulation in Dynamic Electromagnetic Scenes Using Pipeline Thread Pool," 10th International Conference on Computational Intelligence and Security (CIS), pp. 770-774, 2014.

[29] S. Sahin, "C-stream: A coroutine-based elastic stream processing engine", Doctoral dissertation, bilkent university, 2015.

[30] J. Mace, P. Bodikz, M. Musuvathiz, R. Fonseca, K. Varadarajanz, " 2DFQ: Two- Dimensional Fair Queueing for Multi-Tenant Cloud Services," In Proceedings of ACM SIGCOMM Conference, ACM, pp. 144-159, 2016.

[31] M. García-Valls, C. Calva-Urrego, A. Juan, A. Alonso, "Adjusting middleware knobs to assess scalability limits of distributed cyber-physical systems," Computer Standards & Interfaces, Volume 51, pp.95-103, 2017.

[32] H. Chen, Q. Wang, B. Palanisamy, P. Xiong, "DCM: Dynamic Concurrency Management for Scaling n-Tier Applications in Cloud," In IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 2097-2104, 2017.